AI Assistant Coding

Assignment 6.3

Name : GOLI NITHYA **HT. No :** 2303A52066 **Batch:**32

Q1. Task: Prompt AI to generate a function that displays all Automorphic

numbers between 1 and 1000 using a for loop.

Instructions:

o Get AI-generated code to list Automorphic numbers using a for loop.

o Analyze the correctness and efficiency of the generated logic.

o Ask AI to regenerate using a while loop and compare both implementations

Task1.py > ...

```python
'''
1.wrte a python function that genartes automorphic numbers within range from 1 to 1000. using for loop
'''
import time


def is_automorphic(num):
    square = num ** 2
    return str(square).endswith(str(num))
def generate_automorphic_numbers(start, end):
    automorphic_numbers = []
    for num in range(start, end + 1):
        if is_automorphic(num):
            automorphic_numbers.append(num)
    return automorphic_numbers
t1 = time.time()
automorphic_numbers = generate_automorphic_numbers(1, 1000)
print("Time taken using for loop:", time.time() - t1)
print("Automorphic numbers between 1 and 1000 are:", automorphic_numbers)

```

```python
'''
2.write a python function that genartes automorphic numbers within range from 1 to 1000. using while loop
'''
def generate_automorphic_numbers_while(start, end):
    automorphic_numbers = []
    num = start
    while num <= end:
        if is_automorphic(num):
            automorphic_numbers.append(num)
        num += 1
    return automorphic_numbers
t2 = time.time()
automorphic_numbers_while = generate_automorphic_numbers_while(1, 1000)
print("Time taken using while loop:", time.time() - t2)
print("Automorphic numbers between 1 and 1000 using while loop are:", automorphic_numbers_while)
```

```
PS D:\Course\AIAC\28-1-2026> & C:/Python314/python.exe d:/Course/AIAC/28-1-2026/Task1.py
  Time taken using for loop: 0.0004050731658935547
  Automorphic numbers between 1 and 1000 are: [1, 5, 6, 25, 76, 376, 625]
  Time taken using while loop: 0.0004239082336425781
  Automorphic numbers between 1 and 1000 using while loop are: [1, 5, 6, 25, 76, 376, 625]
PS D:\Course\AIAC\28-1-2026> 
```

Explanation:

A for loop usually takes less time and is best used when you know the stopping condition in advance. If the stopping point is not known, a while loop is more suitable.

Q2. Task: Ask AI to write nested if-elif-else conditions to classify online

shopping feedback as Positive, Neutral, or Negative based on a

numerical rating (1–5).

• Instructions:

o Generate initial code using nested if-elif-else.

o Analyze correctness and readability.

o Ask AI to rewrite using dictionary-based or match-case

structure..

Task2.py > classify_feedback_dict

```python
import time
#write a python functuion that classify online shopping feedback into positive, negative and neutral based on numerical rating (1-5) using nested if-elif-else statements.
def classify_feedback(rating):
    if rating >= 1 and rating <= 5:
        if rating >= 4:
            return "Positive"
        elif rating == 3:
            return "Neutral"
        else:
            return "Negative"
    else:
        return "Invalid rating. Please provide a rating between 1 and 5."
# Example usage:
t1 = time.time()
print(classify_feedback(5))
print("Time taken usning if-elif-else statements:", time.time() - t1)  # Output: Positive
print(classify_feedback(3))  # Output: Neutral
print(classify_feedback(1))  # Output: Negative
print(classify_feedback(6))  # Output: Invalid rating. Please provide a rating between 1 and 5.
```

```python
print("\n")
#write a python functuion that classify online shopping feedback into positive, negative and neutral based on numerical rating (1-5) using dictionary or match case.
def classify_feedback_dict(rating):
    feedback_dict = {
        5: "Positive",
        4: "Positive",
        3: "Neutral",
        2: "Negative",
        1: "Negative"
    }
    return feedback_dict.get(rating, "Invalid rating. Please provide a rating between 1 and 5.")
# Example usage:
t2 = time.time()
print(classify_feedback_dict(5))
print("Time taken using dictionary or match case:", time.time() - t2)  # Output: Positive
print(classify_feedback_dict(3))  # Output: Neutral
print(classify_feedback_dict(1))  # Output: Negative
print(classify_feedback_dict(6))  # Output: Invalid rating. Please provide a rating between 1 and 5.
```

```
PS D:\Course\AIAC\28-1-2026> & C:/Python314/python.exe d:/Course/AIAC/28-1-2026/Task2.py
Positive
Time taken usning if-elif-else statements: 0.00010204315185546875
Neutral
Negative
Invalid rating. Please provide a rating between 1 and 5.


Positive
Time taken using dictionary or match case: 3.743171691894531e-05
Neutral
Negative
Invalid rating. Please provide a rating between 1 and 5.
PS D:\Course\AIAC\28-1-2026>
```

Explanation:

Conditional statements help maintain code correctness, and using dictionaries enhances code readability

Q3. Task 3: Statistical_operations

Define a function named statistical_operations(tuple_num) that performs the following statistical operations on a tuple of numbers:

- **Minimum, Maximum**
- **Mean, Median, Mode**
- **Variance, Standard Deviation**

While writing the function, observe the code suggestions provided by GitHub Copilot. Make decisions to accept, reject, or modify the suggestions based on their relevance and correctness

Task3.py ● Task2.py

28-1-2026 > Task3.py > ...

```python
'''
1. write a python function nsame statistical_opeeration(tuple_numm) and perform Minimum, maximum, mean,
median, mode, variance and standard deviation.
'''
import statistics
def statistical_operation(tuple_numm):
    if len(tuple_numm) == 0:
        return "The tuple is empty. Please provide a tuple with numerical values."

    minimum = min(tuple_numm)
    maximum = max(tuple_numm)
    mean = statistics.mean(tuple_numm)
    median = statistics.median(tuple_numm)
    try:
        mode = statistics.mode(tuple_numm)
    except statistics.StatisticsError:
        mode = "No unique mode found"
    variance = statistics.variance(tuple_numm)
    std_deviation = statistics.stdev(tuple_numm)

    return {
        "Minimum": minimum,
        "Maximum": maximum,
        "Mean": mean,
        "Median": median,
        "Mode": mode,
        "Variance": variance,
        "Standard Deviation": std_deviation
    }
# Example usage:
data = (1, 2, 2, 3, 4, 5, 5, 5)
result = statistical_operation(data)
for key, value in result.items():
    print(f"{key}: {value}")
```

```
PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/Activate.ps1
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/28-1-2026/Task3.py
Minimum: 1
Maximum: 5
Mean: 3.375
Median: 3.5
Mode: 5
Variance: 2.5535714285714284
Standard Deviation: 1.5979898086569353
(AIAC_env) PS D:\Course\AIAC>
```

Q4. Task 4: Teacher Profile

• **Prompt: Create a class Teacher with attributes teacher_id, name, subject, and experience. Add a method to display teacher details.**

• **Expected Output: Class with initializer, method, and object creation.**

Task4.py ×

28-1-2026 > Task4.py > ...

```python
'''
create a class with Teacher  with attributes teacher_id, name, subject, and experience
Method to display teacher details
'''
class Teacher:
    def __init__(self, teacher_id, name, subject, experience):
        self.teacher_id = teacher_id
        self.name = name
        self.subject = subject
        self.experience = experience

    def display_details(self):
        print(f"Teacher ID: {self.teacher_id}")
        print(f"Name: {self.name}")
        print(f"Subject: {self.subject}")
        print(f"Experience: {self.experience} years")
# Example usage
teacher1 = Teacher(1, "Alice Smith", "Mathematics", 10)
teacher1.display_details()
teacher2 = Teacher(2, "Bob Johnson", "Science", 8)
teacher2.display_details()
teacher3 = Teacher(3, "Charlie Brown", "History", 5)
teacher3.display_details()
teacher4 = Teacher(4, "Diana Prince", "English", 12)
teacher4.display_details()
```

```
PROBLEMS    OUTPUT    PORTS    DEBUG CONSOLE    TERMINAL

(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/28-1-2026/Task4.py
Teacher ID: 1
Name: Alice Smith
Subject: Mathematics
Experience: 10 years
Teacher ID: 2
Name: Bob Johnson
Subject: Science
Experience: 8 years
Teacher ID: 3
Name: Charlie Brown
Subject: History
Experience: 5 years
Teacher ID: 4
Name: Diana Prince
Subject: English
Experience: 12 years
(AIAC_env) PS D:\Course\AIAC>
```

Q5. Task #5 – Zero-Shot Prompting with Conditional Validation

Use zero-shot prompting to instruct an AI tool to generate a function

that validates an Indian mobile number.

Requirements

• The function must ensure the mobile number:

o Starts with 6, 7, 8, or 9

o Contains exactly 10 digits

Expected Output

• A valid Python function that performs all required validations

without using any input-output examples in the prompt

```
Task5.py ×
28-1-2026 > Task5.py > ...
'''
write a python function name vaild indian number and check number starts with 6 7 8 9
and length is 10 digit
'''
def vaild_indian_number(number):
    if len(number) == 10 and number[0] in '6789' and number.isdigit():
        return True
    return False
print(vaild_indian_number("9876543210"))  # True
print(vaild_indian_number("1234567890"))  # False
```

```
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/28-1-2026/Task5.py
True
False
(AIAC_env) PS D:\Course\AIAC>
```

Q6. Task Description #6 (Loops – Armstrong Numbers in a Range)

Task: Write a function using AI that finds all Armstrong numbers in a user-

specified range (e.g., 1 to 1000).

Instructions:

- **Use a for loop and digit power logic.**
- **Validate correctness by checking known Armstrong numbers (153, 370,**

etc.).

- **Ask AI to regenerate an optimized version (using list comprehensions).**

Expected Output #7:

- **Python program listing Armstrong numbers in the range.**
- **Optimized version with explanation**

28-1-2026 > Task6.py > ...

```python
'''
write a python function that finds armstrong numbers from 1 to 1000
using for loop and digit power logic.
'''
def find_armstrong_numbers():
    armstrong_numbers = []
    for num in range(1, 1001):
        # Convert number to string to easily iterate over digits
        digits = str(num)
        num_digits = len(digits)
        sum_of_powers = sum(int(digit) ** num_digits for digit in digits)

        if sum_of_powers == num:
            armstrong_numbers.append(num)

    return armstrong_numbers
# Example usage
armstrong_numbers = find_armstrong_numbers()
print("Armstrong numbers from 1 to 1000:", armstrong_numbers)
```

```python
'''
write a python function that finds armstrong numbers from 1 to 1000
using for loop and digit power logic. optimized version using list comprehension
'''
def find_armstrong_numbers_optimized():
    return [num for num in range(1, 1001) if sum(int(digit) ** len(str(num)) for digit in str(num)) == num]
# Example usage
armstrong_numbers_optimized = find_armstrong_numbers_optimized()
print("Armstrong numbers from 1 to 1000 (optimized):", armstrong_numbers_optimized)
```

Output:

PROBLEMS OUTPUT PORTS DEBUG CONSOLE TERMINAL

```
PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/Activate.ps1
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/28-1-2026/Task6.py
Armstrong numbers from 1 to 1000: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Armstrong numbers from 1 to 1000 (optimized): [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
(AIAC_env) PS D:\Course\AIAC>
```

Explanation:

`For` loops are easier to understand and debug, making them suitable for beginners, whereas list comprehensions are minimal but difficult to debug

Q7. Task Description #7 (Loops – Happy Numbers in a Range)

Task: Generate a function using AI that displays all Happy Numbers within a

user-specified range (e.g., 1 to 500).

Instructions:

• Implement the logic using a loop: repeatedly replace a number with the

sum of the squares of its digits until the result is either 1 (Happy

Number) or enters a cycle (Not Happy).

• Validate correctness by checking known Happy Numbers (e.g., 1, 7, 10,

13, 19, 23, 28…).

• Ask AI to regenerate an optimized version (e.g., by using a set to detect

cycles instead of infinite loops).

Expected Output #8:

• Python program that prints all Happy Numbers within a range.

• Optimized version using cycle detection with explanation.

```python
write a python function  to display all happnumbers in rnage 1 to 500
'''
import time


def is_happy_number(n):
    seen = set()
    while n != 1 and n not in seen:
        seen.add(n)
        n = sum(int(digit) ** 2 for digit in str(n))
    return n == 1
def happy_numbers_in_range(start, end):
    happy_numbers = []
    for num in range(start, end + 1):
        if is_happy_number(num):
            happy_numbers.append(num)
    return happy_numbers
t1 = time.time()
happy_numbers = happy_numbers_in_range(1, 500)
t2 = time.time()
print("Time taken without optimization:", t2 - t1)
print("Happy numbers between 1 and 500 are:", happy_numbers)
```

```python
'''
write a python function  to display all happnumbers in rnage 1 to 500
Optimize the above code using cycle detection
'''
import time
def is_happy_number_optimized(n):
    def get_next(number):
        return sum(int(digit) ** 2 for digit in str(number))

    slow = n
    fast = get_next(n)
    while fast != 1 and slow != fast:
        slow = get_next(slow)
        fast = get_next(get_next(fast))
    return fast == 1
def happy_numbers_in_range_optimized(start, end):
    happy_numbers = []
    for num in range(start, end + 1):
        if is_happy_number_optimized(num):
            happy_numbers.append(num)
    return happy_numbers
t1 = time.time()
happy_numbers_optimized = happy_numbers_in_range_optimized(1, 500)
t2 = time.time()
print("Time taken with optimization:", t2 - t1)
print("Happy numbers between 1 and 500 are (optimized):", happy_numbers_optimized)
```

Output:

```
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/28-1-2026/Task7.py
Time taken without optimization: 0.0021097660064697266
Happy numbers between 1 and 500 are: [1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100, 103, 109, 129, 130, 133,
 139, 167, 176, 188, 190, 192, 193, 203, 208, 219, 226, 230, 236, 239, 262, 263, 280, 291, 293, 301, 302, 310, 313, 319, 320, 326, 329, 331,
338, 356, 362, 365, 367, 368, 376, 379, 383, 386, 391, 392, 397, 404, 409, 440, 446, 464, 469, 478, 487, 490, 496]
Time taken with optimization: 0.003709077835083008
Happy numbers between 1 and 500 are (optimized): [1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100, 103, 109, 12
9, 130, 133, 139, 167, 176, 188, 190, 192, 193, 203, 208, 219, 226, 230, 236, 239, 262, 263, 280, 291, 293, 301, 302, 310, 313, 319, 320, 326
, 329, 331, 338, 356, 362, 365, 367, 368, 376, 379, 383, 386, 391, 392, 397, 404, 409, 440, 446, 464, 469, 478, 487, 490, 496]
(AIAC_env) PS D:\Course\AIAC>
```

Explanation:

The optimized cycle detection method is better because it uses less memory, runs faster, and works well for large inputs. The set-based method is easier for beginners, but the optimized method is more professional and performance-focused

---

Q8. Task Description #8 (Loops – Strong Numbers in a Range)

Task: Generate a function using AI that displays all Strong Numbers (sum of

factorial of digits equals the number, e.g., 145 = 1!+4!+5!) within a given range.

Instructions:

- **Use loops to extract digits and calculate factorials.**
- **Validate with examples (1, 2, 145).**
- **Ask AI to regenerate an optimized version (precompute digit factorials).**

Expected Output #9:

- **Python program that lists Strong Numbers.**
- **Optimized version with explanation.**

Task8.py

28-1-2026 > Task8.py > ...

```python
'''
write a python function to display strong numbers with given range.
'''
import time


def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
def is_strong_number(num):
    sum_of_factorials = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        sum_of_factorials += factorial(digit)
        temp //= 10
    return sum_of_factorials == num
def display_strong_numbers(start, end):
    strong_numbers = []
    for num in range(start, end + 1):
        if is_strong_number(num):
            strong_numbers.append(num)
    return strong_numbers
# Example usage:
start_range = 1
end_range = 500
t1=time.time()
strong_numbers_in_range = display_strong_numbers(start_range, end_range)
t2=time.time()
print(f"Time taken: {t2-t1} seconds")
print(f"Strong numbers between {start_range} and {end_range}: {strong_numbers_in_range}")
```

```python
'''
write a python function to display strong numbers with given range.
optimized version precomputing factorials.
'''

import time
def precompute_factorials():
    factorials = {}
    for i in range(10):
        factorials[i] = factorial(i)
    return factorials
def is_strong_number_optimized(num, factorials):
    sum_of_factorials = 0
    temp = num
    while temp > 0:
        digit = temp % 10
        sum_of_factorials += factorials[digit]
        temp //= 10
    return sum_of_factorials == num
def display_strong_numbers_optimized(start, end):
    factorials = precompute_factorials()
    strong_numbers = []
    for num in range(start, end + 1):
        if is_strong_number_optimized(num, factorials):
            strong_numbers.append(num)
    return strong_numbers
# Example usage:
start_range = 1
end_range = 500
t1=time.time()
strong_numbers_in_range_optimized = display_strong_numbers_optimized(start_range, end_range)
t2=time.time()
print(f"Time taken (optimized): {t2-t1} seconds")
print(f"Strong numbers between {start_range} and {end_range} (optimized): {strong_numbers_in_range_optimized}")
```

Output:

```
PROBLEMS   OUTPUT   PORTS   DEBUG CONSOLE   TERMINAL

PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/Activate.ps1
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/28-1-2026/Task8.py
Time taken: 0.0006704330444335938 seconds
Strong numbers between 1 and 500: [1, 2, 145]
Time taken (optimized): 0.00022077560424804688 seconds
Strong numbers between 1 and 500 (optimized): [1, 2, 145]
(AIAC_env) PS D:\Course\AIAC>
```

Explanation:

The basic approach recalculates factorials repeatedly, making it inefficient. The optimized approach precomputes digit factorials, reducing computations and improving performance for large ranges

---

Q9. Task #9 – Few-Shot Prompting for Nested Dictionary Extraction

Objective

Use few-shot prompting (2–3 examples) to instruct the AI to create a

function that parses a nested dictionary representing student

information.

Requirements

• The function should extract and return:

o Full Name

o Branch

o SGPA

Expected Output

A reusable Python function that correctly navigates and extracts values

from nested dictionaries based on the provided examples

```
'''
write a python function that parses a nested dictionary of student
information Requirements:
Full Name
o Branch
o SGPA

example input:
 "full_name": "Alice Johnson",
            "branch": "Computer Science",
            "sgpa": 8.5
        full_name": "Bob Smith",
            "branch": "Mechanical Engineering",
            "sgpa": 7.8
'''
```

```python
def parse_student_info(students):
    parsed_info = []
    for student_id, info in students.items():
        student_data = {
            "Full Name": info.get("full_name"),
            "Branch": info.get("branch"),
            "SGPA": info.get("sgpa")
        }
        parsed_info.append(student_data)
    return parsed_info
# Example usage
students = {
    "student_1": {
        "full_name": "Alice Johnson",
        "branch": "Computer Science",
        "sgpa": 8.5
    },
    "student_2": {
        "full_name": "Bob Smith",
        "branch": "Mechanical Engineering",
        "sgpa": 7.8
    },
    "student_3": {
        "full_name": "Charlie Brown",
        "branch": "Electrical Engineering",
        "sgpa": 9.1
        }
}
parsed_students = parse_student_info(students)
for student in parsed_students:
    print(student)
```

Output:

```
PROBLEMS    OUTPUT    PORTS    DEBUG CONSOLE    TERMINAL

(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/28-1-2026/Task9.py
{'Full Name': 'Alice Johnson', 'Branch': 'Computer Science', 'SGPA': 8.5}
{'Full Name': 'Bob Smith', 'Branch': 'Mechanical Engineering', 'SGPA': 7.8}
{'Full Name': 'Charlie Brown', 'Branch': 'Electrical Engineering', 'SGPA': 9.1}
(AIAC_env) PS D:\Course\AIAC>
```

Q10. Task Description #10 (Loops – Perfect Numbers in a Range)

Task: Generate a function using AI that displays all Perfect Numbers within a

user-specified range (e.g., 1 to 1000).

Instructions:

• A Perfect Number is a positive integer equal to the sum of its proper

divisors (excluding itself).

o Example: 6 = 1 + 2 + 3, 28 = 1 + 2 + 4 + 7 + 14.

• Use a for loop to find divisors of each number in the range.

• Validate correctness with known Perfect Numbers (6, 28, 496…).

• Ask AI to regenerate an optimized version (using divisor check only up to

√n)

28-1-2026 > Task10.py > ...

```
'''
write a python funcyion that displays all perfect numbers in range from 1 to 1000
'''
import time
def is_perfect_number(n):
    sum_of_divisors = 0
    for i in range(1, n):
        if n % i == 0:
            sum_of_divisors += i
    return sum_of_divisors == n
def display_perfect_numbers():
    perfect_numbers = []
    for num in range(1, 1001):
        if is_perfect_number(num):
            perfect_numbers.append(num)
    return perfect_numbers
if __name__ == "__main__":
    t1 = time.time()
    perfect_numbers = display_perfect_numbers()
    t2 = time.time()
    print("Time taken to find perfect numbers:", t2 - t1, "seconds")
    print("Perfect numbers between 1 and 1000 are:", perfect_numbers)
```

```
28-1-2026 > Task10.py > ...
'''
write a python funcyion that displays all perfect numbers in range from 1 to 1000 optimized using divisor check only up to square root of n
'''
import time
import math
def is_perfect_number_optimized(n):
    if n < 2:
        return False
    sum_of_divisors = 1  # 1 is a divisor of all n > 1
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            sum_of_divisors += i
            if i != n // i:
                sum_of_divisors += n // i
    return sum_of_divisors == n
def display_perfect_numbers_optimized():
    perfect_numbers = []
    for num in range(1, 1001):
        if is_perfect_number_optimized(num):
            perfect_numbers.append(num)
    return perfect_numbers
if __name__ == "__main__":
    t1 = time.time()
    perfect_numbers_optimized = display_perfect_numbers_optimized()
    t2 = time.time()
    print("Time taken to find perfect numbers (optimized):", t2 - t1, "seconds")
    print("Perfect numbers between 1 and 1000 (optimized) are:", perfect_numbers_optimized)
```

Output:

```
PROBLEMS    OUTPUT    PORTS    DEBUG CONSOLE    TERMINAL

PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/Activate.ps1
(AIAC_env) PS D:\Course\AIAC> & D:/Course/AIAC/AIAC_env/Scripts/python.exe d:/Course/AIAC/28-1-2026/Task10.py
Time taken to find perfect numbers: 0.021500110626220703 seconds
Perfect numbers between 1 and 1000 are: [6, 28, 496]
Time taken to find perfect numbers (optimized): 0.001857757568359375 seconds
Perfect numbers between 1 and 1000 (optimized) are: [6, 28, 496]
(AIAC_env) PS D:\Course\AIAC>
```

Explanation:

The basic method checks all divisors up to the number, leading to higher time complexity. The optimized method checks divisors only up to the square root, improving efficiency while giving the same correct results